How To Receive Email in a Rails App

Filed under
email
rails
ruby

The number one article suggestion thus far is how to receive email in a rails app. Ask and you shall receive. What I am about to present is just a first run at it and I am not going to promise that it scales. :) That said, I fooled around with it a bit and found it relatively easy. I will post more on the topic as I play more with receiving email, but what follows is enough to at least get you up and running. The email address I used for this is actually a Gmail address, so using this method does not involve postfix or some other mail server configuration.

The Project

The project that I added email receiving functionality to was YardVote, a weekend project by my friends at Collective Idea. YardVote allows people to record a "yard" with the political signs that were present in said yard and displays them on a google map. I was immediately enamored with the idea but found entering signs on my iPhone to be a bit tedious. Giving that everyone seems to be interested in receiving email, that it would be handy for yardvote, and that yardvote was open source, I decided to take a crack at it.

The Process

The path to success in summary form was this:

1. Check for new emails
2. Create new or update existing location from each email
3. Archive emails that have been processed to avoid duplicates
4. Turn process of checking and processing email into daemon

bin/mail_receiver.rb

For whatever reason, I created a bin directory and a mail_receiver.rb script inside of that. Actually, it was for a reason. I think bin seems like a good place for stuff like this. I could have used the script directory, but I figured I would leave that for Rails and plugins. I will over comment the code below to help with any parts of it that may raise questions.

```
# default rails environment to development
ENV['RAILS_ENV'] ||= 'development'
# require rails environment file which basically "boots" up rails fo
require File.join(File.dirname(__FILE__), '..', 'config', 'environme
require 'net/imap'
require 'net/http'

# amount of time to sleep after each loop below
SLEEP_TIME = 60

# mail.yml is the imap config for the email account (ie: username, h
config = YAML.load(File.read(File.join(RAILS_ROOT, 'config', 'mail.y

# this script will continue running forever
loop do
  begin
    # make a connection to imap account
    imap = Net::IMAP.new(config['host'], config['port'], true)
    imap.login(config['username'], config['password'])
    # select inbox as our mailbox to process
    imap.select('Inbox')

    # get all emails that are in inbox that have not been deleted
    imap.uid_search(["NOT", "DELETED"]).each do |uid|
      # fetches the straight up source of the email for tmail to pars
      source   = imap.uid_fetch(uid, ['RFC822']).first.attr['RFC822']
```

```
      # Location#new_from_email accepts the source and creates new lo
      location = Location.new_from_email(source)

      # check for an existing location that matches the one created f
      existing = Location.existing_address(location)

      if existing
        # location exists so update the sign color to the emailed loc
        existing.signs = location.signs
        if existing.save
          # existing location was updated
        else
          # existing location was invalid
        end
      elsif location.save
        # emailed location was valid and created
      else
        # emailed location was invalid
      end

      # there isn't move in imap so we copy to new mailbox and then d
      imap.uid_copy(uid, "[Gmail]/All Mail")
      imap.uid_store(uid, "+FLAGS", [:Deleted])
    end

    # expunge removes the deleted emails
    imap.expunge
    imap.logout
    imap.disconnect

  # NoResponseError and ByResponseError happen often when imap'ing
  rescue Net::IMAP::NoResponseError => e
    # send to log file, db, or email
  rescue Net::IMAP::ByeResponseError => e
    # send to log file, db, or email
  rescue => e
    # send to log file, db, or email
  end

  # sleep for SLEEP_TIME and then do it all over again
  sleep(SLEEP_TIME)
end
```

Location#new_from_email

The only piece of code that you might need to help what I showed above make sense is the `Location#new_from_email` method.

```
class Location < ActiveRecord::Base
  def self.new_from_email(source)
    attrs, email = {}, TMail::Mail.parse(source)
    # set signs attribute equal to subject in proper form
    attrs[:signs]   = email.subject.blank? ? '' : email.subject.downc
    # set street equal to the body with email signatures stripped
    attrs[:street]  = parse_address(email.body)
    # create new location from the attributes
    new(attrs)
  end

  def self.parse_address(body)
    body.split("\n\n").first
  end
end
```

Location#new_from_email Specs

While working on the `new_from_email` method, I created an emails directory inside fixtures that had several ways an email could be sent to YardVote. Then, I created a few specs to make sure that `new_from_email` was actually working.

```
describe Location do
  describe "#new_from_email" do
    it "should be subject" do
      location = Location.new_from_email(email_fixture(:red_no_subjec
      location.should have_error_on(:signs)
    end

    it "should require body" do
      location = Location.new_from_email(email_fixture(:red_no_body))
      location.valid?
      location.errors.full_messages.should include("We can't find a p
    end
```

```
    it "should set street to email body" do
      location = Location.new_from_email(email_fixture(:red))
      location.street.should == '1600 Pennsylvania Ave. Washington, D
    end

    it "should set sign equal to subject" do
      location = Location.new_from_email(email_fixture(:red))
      location.signs.should == 'Red'
    end

    it "should work with mixed case subject" do
      location = Location.new_from_email(email_fixture(:red_mixedcase
      location.signs.should == 'Red'
    end

    it "should work with upper case subject" do
      location = Location.new_from_email(email_fixture(:red_uppercase
      location.signs.should == 'Red'
    end

    it "should work with multi line body" do
      location = Location.new_from_email(email_fixture(:red_address_t
      location.valid?
      location.to_location.to_s.should == "1600 Pennsylvania Ave Nw\r
    end

    it "should work with multi line body and email signature" do
      location = Location.new_from_email(email_fixture(:red_address_t
      location.valid?
      location.to_location.to_s.should == "1600 Pennsylvania Ave Nw\r
    end
  end
end
```

bin/mail_receiver_ctl.rb

Now that I could check for new email and process that email, it was time to daemonize the script. The benefit of daemonizing is that you get nice and easy start, stop and restart commands, along with a PID. The PID makes monitoring your script possible, which is needed because it is bound to crash or begin sucking up to much memory.

```
require 'rubygems'
require 'daemons'
dir = File.dirname(__FILE__)
Daemons.run(dir + '/mail_receiver.rb')
```

What, you wanted more? Now you can run commands like `ruby bin/mail_receiver_ctl.rb start` to start your script and `ruby bin/mail_receiver_ctl.rb stop` to stop it. Very handy.

config/mail.god

The whole setup was running fine for a while, but sure enough, a few days later, I noticed that a few of my emails were still sitting in the inbox. This was because the script had crashed. I fiddled around with god a little bit and came up with the following script, mostly stolen from Ryan Bates Railscast on god.

```
RAILS_ROOT = File.expand_path(File.join(File.dirname(__FILE__), '..')

God.watch do |w|
  # script that needs to be run to start, stop and restart
  script          = "ruby #{RAILS_ROOT}/bin/mail_receiver_ctl.rb"
  # attaching rails env to each script line to be sure the daemon sta
  rails_env       = "RAILS_ENV=production"

  w.name          = "mail-receiver"
  w.group         = "mail"
  w.interval      = 60.seconds
  w.start         = "#{script} start #{rails_env}"
  w.restart       = "#{script} restart #{rails_env}"
  w.stop          = "#{script} stop #{rails_env}"
  w.start_grace   = 20.seconds
  w.restart_grace = 20.seconds
  w.pid_file      = "#{RAILS_ROOT}/log/mail_receiver.pid"

  w.behavior(:clean_pid_file)

  w.start_if do |start|
    start.condition(:process_running) do |c|
      c.interval = 10.seconds
      c.running = false
```

```
    end
  end

  w.restart_if do |restart|
    restart.condition(:memory_usage) do |c|
      c.above = 100.megabytes
      c.times = [3, 5]
    end

    restart.condition(:cpu_usage) do |c|
      c.above = 80.percent
      c.times = 5
    end
  end

  w.lifecycle do |on|
    on.condition(:flapping) do |c|
      c.to_state = [:start, :restart]
      c.times = 5
      c.within = 5.minute
      c.transition = :unmonitored
      c.retry_in = 10.minutes
      c.retry_times = 5
      c.retry_within = 2.hours
    end
  end
end
```

Conclusion

Two hours, three files and a few extra methods later, YardVote could receive email. Before too long, I will be adding custom email addresses per person into an application such as Flickr and Highrise. I will be sure to post on that here, but, for now, I hope this helps people get started. Also, you can see the code actually implemented in the app on github.

I think we need to get more knowledge share on how to do things like this in Rails. If you have added receiving emails into your Rails app, how did you do it? Post a comment or link to an article that you wrote or used. Below are a few of the articles I have seen around the interwebs.

Receiving Email in Rails links

- Stress-free Incoming E-Mail Processing with Rails
- respond_to.email, or how to handle incoming emails in rails RESTfully
- Email on rails

via railstips.org

The basic functionality is in the post, but the most useful information on running a scalable process is in the comments.